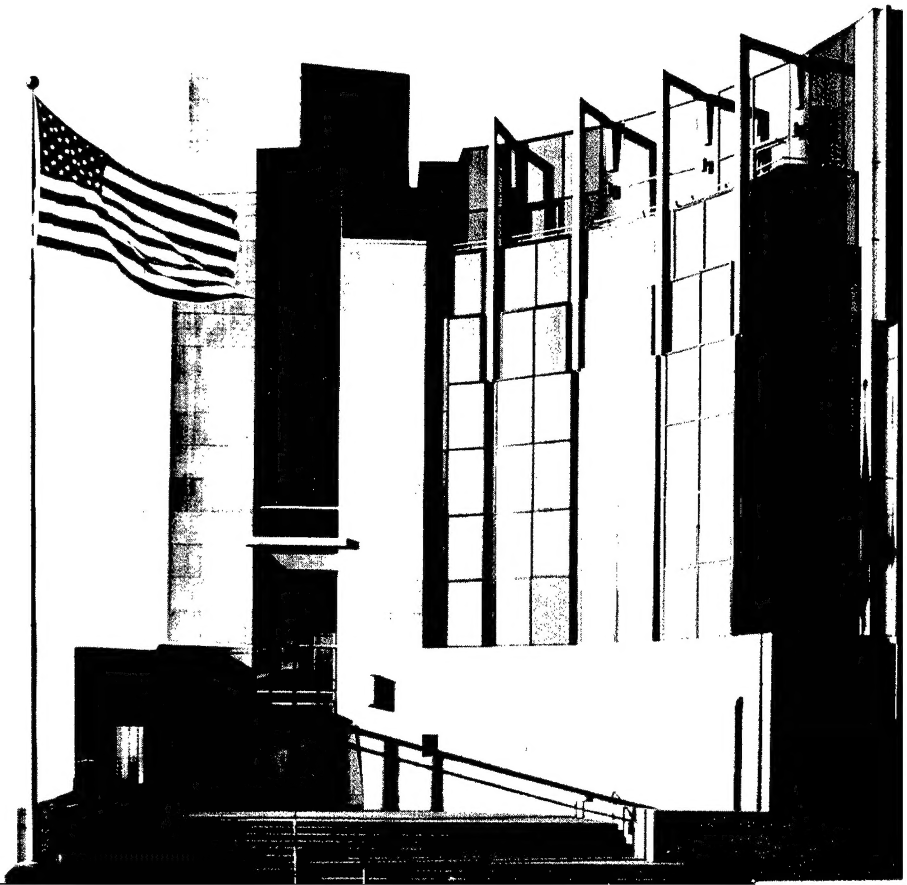
Carnegie Mellon
Software Engineering Institute

# A Basis for an Assembly Process for COTS-Based Systems (APCS)

David J. Carney
Patricia A. Oberndorf
Patrick R.H. Place

*May 2003*



TECHNICAL REPORT
CMU/SEI-2003-TR-010
ESC-TR-2003-010

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

# A Basis for an Assembly Process for COTS-Based Systems (APCS)

CMU/SEI-2003-TR-010
ESC-TR-2003-010

David J. Carney
Patricia A. Oberndorf
Patrick R.H. Place

*May 2003*

**COTS-Based Systems**

AQM03-08-2102

# Table of Contents

**Acknowledgments** ........ v

**Abstract** ........ vii

**1 Introduction** ........ 1

**2 Conceptual Basis of the Process** ........ 3
- 2.1 Requirements ........ 3
  - 2.1.1 Differing Approaches to Requirements ........ 3
  - 2.1.2 Independence of the Marketplace ........ 4
  - 2.1.3 Negotiables and Non-Negotiables ........ 5
- 2.2 Spiral Development ........ 6
- 2.3 The Need for Knowledge ........ 7
  - 2.3.1 The Body of Knowledge and the System View ........ 8
- 2.4 The Role of Risk ........ 10
- 2.5 Relation to Other Work ........ 11

**3 The APCS Process** ........ 13
- 3.1 Guiding Principles ........ 13
- 3.2 Process Overview ........ 14
  - 3.2.1 Other Classes of Activities ........ 14
  - 3.2.2 Discovery ........ 15
  - 3.2.3 Assembly ........ 17
  - 3.2.4 Evaluation and Assessment ........ 18
- 3.3 Ongoing Iterations ........ 19
- 3.4 System Deployment ........ 20

**4 A Model of APCS** ........ 21
- 4.1 Manage Program and Assess Iteration ........ 22
- 4.2 Perform Iteration ........ 22
  - 4.2.1 Step 1: Plan Iteration ........ 24
  - 4.2.2 Step 2: Construct System ........ 26
  - 4.2.3 Step 3: Evaluate System ........ 30

**5 Summary and Conclusion** .......... **31**
5.1 Instantiation .......... 31
5.2 Further Work .......... 32

**References/Bibliography** .......... **33**

# List of Figures

Figure 1: The Traditional Placement of Requirements ........................................ 4

Figure 2: Competing "Centers of Gravity" for a COTS-Based System .................... 5

Figure 3: Classes of Knowledge ................................................................ 8

Figure 4: Discovery Activities ................................................................ 17

Figure 5: Ongoing Iterations of APCS .......................................................... 19

Figure 6: Top-Level View of APCS .............................................................. 21

Figure 7: Perform Iteration .................................................................... 23

Figure 8: Detail of Perform Iteration .......................................................... 23

Figure 9: Further Decomposition of Plan Iteration ............................................ 25

Figure 10: Further Decomposition of Construct System ........................................ 26

Figure 11: Low-Level Activities of Refine Data ............................................... 28

# Acknowledgments

The work discussed in this report owes a great deal to both the work of the SEI's COTS-Based Systems Initiative and Barry Boehm's spiral model. The authors are grateful for the contributions from these sources.

We are also particularly grateful to Ceci Albert, Lisa Brownsword, Fred Long, Mary Jo Staley, and Kurt Wallnau for their comments on earlier drafts of the work described hereafter. Further, we thank Claire Dixon for her work in editing the document.

# Abstract

This paper describes a generic process framework for developing software systems based on commercial off-the-shelf (COTS) products. The framework is based on Barry Boehm's familiar spiral development process. However, it is primarily intended for projects that make significant use of commercial components and other pre-existing software as elements of the system to be fielded. The aspects of the process that are most affected by this reliance on COTS components lie in the area of requirements, and the description of the process is most extensive in that area. The necessity of using system prototypes as the major vehicle for reducing risk is assumed, as are parallel and interleaved periods of gathering and refining knowledge about the system to be built. Each element of the process is first described and then depicted in several models, using Integrated Definition modeling technique (IDEF0). The paper describes how the interactions between the candidate COTS components, the stakeholders' implicit and explicit needs, and the context in which the system will operate all provide interacting constraints on both the process and the resulting system.

# 1 Introduction

The major complicating factor for building and maintaining software systems today, where few new systems are built to order, and where commercial products are expected to play a major role, is the diverse nature of the things that influence a new system's shape and functioning. To be sure, the needs of a system's stakeholders are critical for shaping the system, as they have always been. But now, those needs are in competition with several other shaping forces: characteristics of the available commercial off-the-shelf (COTS) products,[1] the degree of necessary interactions with legacy systems, long-term risks stemming from marketplace reliance, and numerous other such factors that determine how a system is eventually built. This richness of constraint impels many changes in system development methods, particularly on requirements gathering and system design, as well as a greatly increased need for product evaluation. In a commercially driven mode of software development, we assert that an iterative approach is an imperative rather than a merely beneficial practice. In addition, as was true in Boehm's spiral model, the conceptual foundation of the work that we present here is that COTS-based software development should be rooted in prototyping and risk reduction; a software system will grow through several iterations to gradually arrive at ("spiral" toward) a fully functional system [Boehm 88].

In this paper, we describe an assembly process for COTS-based systems (APCS), a generic process for developing COTS-based software systems that can be instantiated in a number of ways.[2] We caution that we do not provide in this paper a lengthy discussion of the complex set of difficulties that COTS-based systems pose for the developer; these have been exhaustively described in many other sources. The reader who desires such information should consult <http://www.sei.cmu.edu/cbs/monographs.html> as a starting point for information on general issues relating to COTS-based systems.

1 Not all commercial products are used "off-the-shelf," and not all off-the-shelf products are truly commercial. Nonetheless, we will continue to use the convenient acronym of COTS ("commercial off-the-shelf"), though only in its broadest sense, since the nuances of what "commercial" really means, or the distinctions between "off-the-shelf" and products that are in some way modified, are not the central topic of this paper. See Carney for a lengthy explication of these issues [Carney 01].

2 In truth, a term such as "framework" might better describe these concepts. However, "framework," like many other vogue words current today, tends to obscure rather than clarify, and we choose instead to describe APCS as a process, albeit one that requires instantiation to be used. Later in this paper we describe one instantiation that has already been put into practice.

The process we describe is based on experience: it is based on extensive analysis of numerous programs, both successful and otherwise, that made significant use of COTS and other classes of pre-existing software. These experiences were drawn from several sources: defense programs, programs from other government agencies, and programs executed entirely by and for industry organizations. These programs have concerned systems ranging from the very small to the very large. We have aimed, therefore, to describe a process that has sufficient generality to be applicable to almost any of those programs. We caution that this work is independent of the SEI's more familiar work in capability maturity models: the process described here is independent of any placement, high or low, on any maturity scale.

While our focus is on the activities of assembling a COTS-based system, it cannot be doubted that many other activities—management, procurement, contracting—are no less affected by COTS products and the COTS marketplace. However, a description of these activities is not pertinent to the engineering activities of Assembly itself. Other work may address this issue in the future.

To prepare the reader for the actual description of the process, we found it necessary to provide some intellectual context, and Section 2 is therefore a somewhat lengthy discussion of the conceptual background upon which the process is based. In Section 3 we define a set of principles that govern the process and describe it at a relatively high level. In Section 4 we describe the activities of the process by means of a set of models that decompose the process into several levels of detail. Section 5 is a conclusion that summarizes the process.

# 2 Conceptual Basis of the Process

There are four issues that warrant discussion: (1) requirements; (2) spiral development; (3) the expanded need for knowledge in building COTS-based systems; and (4) the role of risk in the APCS process.

## 2.1 Requirements

The matter of requirements is primary, as we have already noted. In this section we explain the changes that a COTS-based approach levies on the way requirements are defined, gathered, and used. Many of the specific details of the APCS process are the direct result of these changes.

### 2.1.1 Differing Approaches to Requirements

In a custom-made software system, the role of requirements is generally thought to be fundamental. In a custom development approach such as the "waterfall" process, the requirements define all characteristics of the software as obligatory and necessary; the requirements specification is the driver of all subsequent activities [Boehm 88]. In this familiar model of software engineering, requirements are elicited through consultation with the system's users and stakeholders, to derive a collection of system capabilities (the "requirements") that then guide the development process. While the practice is familiar, however, the tasks of eliciting and managing requirements have proven remarkably difficult, and have been major causes of a large number of failed programs.

In a COTS-based paradigm, requirements are no less central to success, but we believe that the familiar process—"nail down the requirements"—is even more prone to failure. The most obvious flaw is the vain hope that one can set forth some abstract set of necessary features and then expect that a group of COTS products will exist that just happen to meet those needs. Whatever the stakeholders' needs might be, and regardless of how well they are elicited and documented, there will likely be an inherent conflict with the capabilities and attributes of the available COTS products. When added to the existing requirements-centered problems, the COTS-specific aspects of requirements increase by far the complexity of an already difficult task.

The influence of COTS products on requirements is an issue that besets many software developers today, and there is little common wisdom on how to deal with the issue. On one hand, genuine requirements do not disappear: there are certain features and characteristics that must be present in the software systems we acquire. On the other hand, many customers are commonly (though not always happily) accepting COTS-based systems that vary considerably from their stated needs.

### 2.1.2 Independence of the Marketplace

We therefore believe that in embarking on a COTS-based project, those involved must first face the fundamental reality that the COTS software marketplace is an independent entity, with its own logic and imperatives. We can illustrate this changed reality through the metaphor of gravity. In a custom development, we could depict a project as in Figure 1, where the needs of the stakeholders provide a "center of gravity" to the system's requirements:

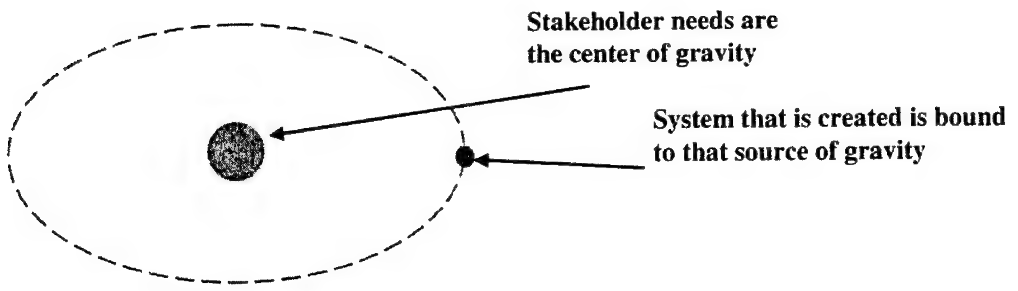

Stakeholder needs are the center of gravity
System that is created is bound to that source of gravity

*Figure 1: The Traditional Placement of Requirements*

By contrast, for a COTS-based system the reality is that shown in Figure 2, where the marketplace forms an equal and opposite "center of gravity," one that is no less a constraint on the system, its stakeholders, and its developers.[3] By "equal and opposite" we mean that COTS software vendors have little motivation to bow to the requirements of a particular software project. Instead, the imperatives that drive COTS software vendors are commercial strategy, market share, and the continual need to stay apace of the technologies their products rely upon.

[3] Even though it is intended simply as a metaphor, Figure 2 has provoked the criticism that such an orbit could never exist. In fact, however, such a system is a specialized instance of the classic "Three Body Problem" from Newtonian mechanics, a complex problem in mathematics. Figure 2 depicts the "*Restricted* Three Body Problem," and the orbit of the small central body can indeed be calculated. The orbit is far more complex than the one shown, but does in fact resemble a complex "figure 8," as the above diagram suggests. See <http://angels.sewanee.edu/Angels/D/doneveu0/html/doneveu0.html>.

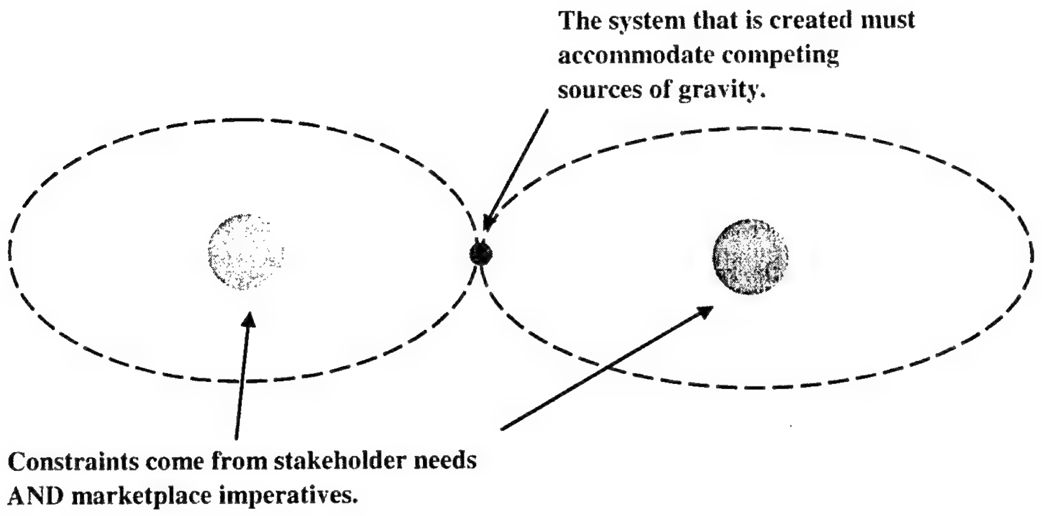

The system that is created must accommodate competing sources of gravity.
Constraints come from stakeholder needs AND marketplace imperatives.

*Figure 2: Competing "Centers of Gravity" for a COTS-Based System*

The exact nature of the resulting "orbit" will depend on the relationships between the two "suns" (e.g., the extent to which the system will depend on COTS products) and will vary from system to system.

### 2.1.3 Negotiables and Non-Negotiables

A COTS-based system must somehow accommodate these competing "sources of gravity." Therefore, to the extent that the features of its constituent COTS products become features of a system, some—perhaps many—of that system's features now lie outside the control of its developers or the stakeholders' stated needs. This forces us to believe that a criterion for what is a genuine requirement must be far more stringent. In practice, this has a profound effect on our perception of what a truly necessary feature should be.

One heuristic is to view the real requirements as those things that spell the difference between making use of a COTS product and using custom development to satisfy a need. As an example, the assertion that "requirement X is utterly and absolutely necessary for the system" implies that either (1) the requirement is already satisfied by a COTS product (or can be satisfied in some effective and inexpensive manner), or (2) must be satisfied in the traditional manner, with custom code. In other words, were the system delivered without that requirement, the system would fail its purpose.

By taking this perspective, we force ourselves to consider whether requirement X is really "make-or-break." We thereby segment the candidate requirements—features and characteristics of the system—into the utterly necessary on one hand, and everything else on the other. The former are indeed the "real" requirements, and these must be satisfied by the completed system. But these needs (assuming that one hopes to satisfy them through COTS products) should be a relatively small set of items. A larger group will be a collection of things that we *desire* the system to have, but that are in one way or another optional. These things will be diverse in nature: end-user preferences about functional behavior; programmatics such as optimal schedule and budgetary boundaries; goals about interfaces with multiple legacy systems; and so forth. As the system develops, some of these will be fully satisfied, some satisfied only partially, and some not at all.

This perspective suggests that the builders of the system must perform two very different tasks. The first is distinguishing the absolutes—the true requirements, which many people term "non-negotiable requirements"—from everything else. The second, and probably more difficult task, is dealing with the large and diverse collection of the "everything else"—the desires and preferences that are commonly called "negotiable requirements." Since at least some of these desires and preferences will be fulfilled, then the way we make decisions about which ones to meet and which to ignore, and which stakeholders to satisfy and which to disappoint, becomes a major challenge.[4]

The complex and often tangled nature of these two tasks is, in fact, the prime motivation for the process steps we propose. Requirements are primary as always, yet must be pared to the smallest possible set, thereby increasing the likelihood of finding COTS products that satisfy them. Making the choices about the remaining items—candidate requirements, preferences, however they are termed—depends on many factors. Above all, it depends on good and reliable knowledge about the stakeholders' real needs and the precise ways that specific COTS products do or do not accommodate them. Our belief is that for both of these tasks, the needed knowledge can only come from actual, hands-on evaluation and use; hence we stress a prototyping approach. In fact, the need for knowledge is the basis of our entire process: better knowledge about our expectations for our systems, better knowledge about the context in which those systems will operate, and better awareness of the capability of COTS products to satisfy all of the stakeholders' needs.

## 2.2 Spiral Development

Boehm introduced the notion of "spiral development" as a means of reducing the risk in program development [Boehm 88]. Spiral development differs from the earlier sequence of the "waterfall" paradigm in that it is based on the notion of iteratively developing the system. The spiral approach makes frequent use of prototypes and demands considerable interaction with the system's end users, thus permitting the developers to gradually refine their understanding of the system's goals, mission, and functionality. The spiral approach means that there is no need to determine all of the system requirements before development begins. Each iteration enables the system devel-

[4] Another challenge is terminology itself. We note that the expression "negotiable requirements" is oxymoronic, and the expression "requirement tradeoff" is illogical: something either is or is not required. Yet any attempt to substitute a term like "preference" will fail; "requirement" is too firmly entrenched in the common vocabulary. We shall therefore follow the customary practice, though we shall often use the umbrella term "need" to connote something that may eventually be defined either as a true requirement or as a negotiable item, depending on how events unfold.

opers to refine their knowledge by collecting additional requirements (or providing greater detail to existing requirements) through interaction with the using community. A spiral approach to development, in one form or another, is widely used on many software systems today, and the process we describe below builds on that basis.

The spiral approach, particularly in the context of COTS products, blurs the distinction between development and maintenance. From a purely technical viewpoint, in the spiral approach, maintenance activities are no different from initial development insofar as each spiral involves refinement of requirements (whether existing or new) and development of a resulting executable. Hence, a particular deployed instance of the system may be almost arbitrarily chosen as being the separator between these two activities. Building COTS-based systems reinforces this view (i.e., that development and maintenance spirals are essentially identical), since normal maintenance activities such as updating a subsystem will be forced by the necessity to incorporate COTS product upgrades even during initial development.

Although Boehm did not use the term "iteration" in the original papers on spiral development, common use equates "iteration" with "spiral," a practice that we shall follow.

## 2.3 The Need for Knowledge

A central concept about which the process revolves is a construct we term the "Body of Knowledge." The knowledge we are concerned with stems from a number of different sources. Some of these are familiar to the developer of custom software:

- the explicit needs and desires of the system's stakeholders
- the end-user processes, both the as-is (of any currently existing system) and the to-be (that the new system will support)
- all of the programmatic constraints: available resources, requisite schedule, and so on
- risks and their mitigations
- system architecture
- constraints of the existing legacy systems with which the new system must interact (e.g., protocols, interfaces, data formats)

Other sources of knowledge are less familiar:

- the COTS products that are currently available
- the vendors of these products: their commercial stability, their licensing strategies, their record of customer satisfaction, and so forth
- the technology trends that will drive future COTS products and future evolution of the system

These new sources are not simply additional bits of knowledge. Instead, the new factors may disrupt the old, forcing us to view our familiar knowledge—stakeholder needs, new business processes, and so on—in a new light. For instance, use of some COTS products might be absolutely mandatory (e.g., for economic, schedule, or policy reasons). But their use may place unexpected constraints on a system's architecture. There may be unseen design assumptions that underlie the COTS products and that affect other elements of the system. Characteristics of available products may compel changes to existing business processes solely because of the unchangeable workings of these products. And perhaps most important, the evolving understanding that emerges as we become more familiar with both the products' capabilities and the needs of the stakeholders will interact with and refine all of the other knowledge we have.

We therefore find that all of these diverse pieces of knowledge, both familiar and novel, must be collected and recorded. This is the Body of Knowledge. This knowledge is likely to fall into the classes depicted in Figure 3. If a system is to become a reality, then these must be harmonized into a coherent view that can be realistically implemented.

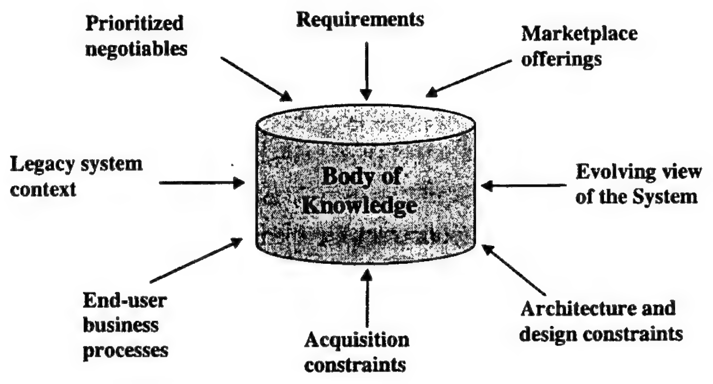


*Figure 3: Classes of Knowledge*

All of this interest in garnering the necessary knowledge leads us to place a period of Discovery—unearthing the needed knowledge—at the head of each iteration.

## The Body of Knowledge and the System View

We describe the Body of Knowledge as the aggregate collection of understanding about the system being built by the organization that builds it; the Body of Knowledge is the sum total of the information learned by the humans who are enacting a given project. It includes technical information, expressions of need from end-users about their daily work, knowledge about COTS products and technology trends, knowledge about the system's operational environment, and knowledge about both the "as-is" and "to-be" business processes. It also includes programmatic information: the potential and actual budgets, data about how the program has been conducted and the nature and effect of bureaucratic or political influences on the program, and so forth.

Traditional software development has always relied on knowledge of this type, although it has seldom been termed a "body of knowledge" as such. Instead, this concept is commonly embedded in a set of artifacts, usually a collection of documents (e.g., System Requirements Specification) that play a role similar to the one we posit here for the Body of Knowledge. In traditional development, the nature and formality of these artifacts has varied widely. Thus, in a project of very small scope, the artifacts' formality may be quite limited. For instance, a project executed by a single person may have no need for formal specification of requirements or heavyweight modeling techniques; the individual's memory may even be sufficient to minimize the amount of information that must be documented. But this is the exception: in most cases (and in any project of substantial scope), the recording, storing, cross-referencing, and retrieving of all project information is utterly critical to the success of the project.

There need not be a large difference, in COTS-based development, from the traditional mechanisms and techniques used for documenting such knowledge. To record knowledge about requirements, for instance, projects have always needed to manage all of the requirements that were elicited, even from the very start of the process, and we have the same need here. The design of a system will be documented in growing technical detail, and this is as true for a COTS-based system as any other.

There will be some differences, however. In traditional development, the primary focus has commonly been on the completed set of requirements; in APCS we maintain that the *full* collection of needs—not yet determined to be negotiable or non-negotiable requirements—must be managed with the same care. Also, we now must keep track of the characteristics of COTS products and their vendors, legacy constraints, and so forth. In general, due to the importance of the marketplace, COTS-based development has a richer set of interrelated data sources, and in many cases there are no traditional document types in which to record them.

There is also a growing awareness that the set of needed artifacts for COTS software development is no longer defined in advance. There is (or should be) a reduced need for a full requirements specification at the outset. The criteria for system acceptance are altered, since there may be a willingness—or even a demand—for deployment of partial functionality. Many organizations now have embraced the notion of Business Process Reengineering, even though there are few standard formats for recording existing business processes.[5] There are many other unfamiliar artifacts: competing lists of negotiable requirements, sensitive documents such as projections about the future health of a vendor, and so forth.

Given both the growth of knowledge and the expanded (and less well-defined) types of artifacts needed, we believe that a COTS-based approach has two related, though distinct needs for all of

[5] There are also few (if any) standard notations to indicate how such processes might be reengineered, and few (if any) standard mechanisms to relate elements of business processes to product capabilities.

this knowledge. One is general: the project personnel need to keep track of all of the wide and diverse information, programmatic as well as technical, that is uncovered as a project progresses.

The second is more focused, since there is a portion of the Body of Knowledge that pertains specifically to the creation of the system prototypes; this portion of the Body of Knowledge we term the "System View." The System View progressively articulates the emerging system concepts and eventually the full system detail from an engineering perspective. It is knowledge that has been converted into some set of artifacts that directly characterize and specify the system under development.

In APCS we do not specify the particular artifacts of the System View (nor how the Body of Knowledge is implemented), since there is a near-infinite set of factors that may constrain or guide those choices: how knowledge is placed under configuration management, choices for document formats, organizational standards, government regulations, and so forth. In one of the many possible instantiations, a project being executed under the constraints of DoD5000.2 has a required collection of documents that must be created and maintained, and these are necessarily part of that project's System View.

Although we do not discuss it here, we believe that there is an analog to the System View, called the Project View, which consists primarily of programmatic knowledge. Just as some of the artifacts of the System View are familiar and some not, the Project View also consists of both familiar artifacts (e.g., an Acquisition Strategy) and unfamiliar ones (e.g., projections about future vendor alliances). Generally speaking, the System View and the Project View are the primary constituents of the Body of Knowledge.

## 2.4 The Role of Risk

We regard the reduction of risk as a central motivation for taking an iterative approach (as was also true in Boehm's original spiral work [Boehm 88]). However, it is often unclear just how this concept is put into practice. Thus, while most organizations now have formal programs in risk management, and government directives often promote a risk-based approach [Clinger-Cohen 96], the bridge between identifying some abstract set of risks and the details of managing a software project through risk reduction (particularly reducing the risks in a COTS-based project) is often unclear.

We suggest a pragmatic and familiar approach to this problem, based on the spiral iterations of the APCS process. In the model descriptions below, we will refer to a set of <u>Iteration Objectives</u> that are formulated for each prototype; the process specifies that these objectives are then transformed into a <u>Detailed Iteration Plan.</u> Our approach is that, as they are created, the objectives should be defined in two ways. The first is a statement of each precise technical or functional goal that the iteration is intended to meet. Such goals could include finding out whether a particular

COTS product will satisfy certain end-users, whether several products can be successfully integrated, whether an interface to a legacy system is possible, and so forth. In addition, however, each individual technical objective should be explicitly linked with a risk to which it relates. This enforces a direct relationship between technical goals and risks. The risk statement should also define whether the technical objective aims at risk avoidance, risk mitigation, or exploring some acceptable level of risk. Using this technique, we wed each technical or functional objective to an unambiguous statement of how that objective reduces or avoids a particular risk.

Note that the *scopes* of the technical/functional objectives and the risks may not be entirely parallel. Thus, some technical objectives are broad-based (e.g., determining whether a particular architecture will meet safety-critical needs) and others relate to details (e.g., placement of icons on a screen). Some risks are such that they jeopardize the entire project; others might encompass only some small subset of stakeholders. This means that there will not necessarily be a one-to-one relationship between technical goals and risks. Several technical objectives might be aimed at mitigating a single risk, and vice-versa.

The explicit relation of risk to technical goal, through the medium of iteration objectives, is a basic element of APCS and other spiral processes, and provides valuable insight for the ongoing iterations of the process. For instance, it may at some point be noticed that several critical risks continue to exist, yet no technical steps are being taken to mitigate them. A realization such as this can be a useful corrective to the definition of objectives for succeeding iterations. By contrast, if some iteration were planned around several technical or functional objectives that mitigate every critical risk facing the program, this may suggest that the planning should be reconsidered and those objectives divided among several iterations; it may also suggest that the risks themselves need to be reexamined.

## 2.5 Relation to Other Work

The development of APCS took place in the context of many other research efforts. By far the greatest debt is to Boehm's spiral model, as well as to his later work in MBASE [Boehm 99]. Our extensions of his work are primarily to focus the spiral model toward the problems of COTS-based software systems. In that sense, our work is less general than the "pure" spiral model, and has little relevance if commercial products do not play a part in a system's construction. Not only is a spiral model the basis for APCS; we believe that it is a necessary basis for *any* COTS-based development process.

APCS was also developed in the larger context of the SEI's work with many aspects of COTS-based systems. Much of that work has been documented in papers, books, and courses of instruction, and a large number of technical papers can be found at <http://www.sei.cmu.edu/cbs/monographs.html>. The intellectual basis of APCS was therefore developed with considerable cross-pollination with different COTS-related projects. There are three of these projects in particular that we shall note. One project was the creation of an extensive tutorial on the evaluation of COTS products;[6] this work is reflected in APCS through our stress on product evaluation and the concomitant need for flexible requirements. The tutorial provides information on various evaluation methods, descriptions of criteria development techniques, and weighting strategies that can be used. It describes an evaluation process whose essential concepts have been preserved in APCS, though with considerably less detail (since we had no desire to re-implement existing work).

A second project was the development, documented in a major textbook, of engineering methods for the design and building of COTS-based systems, entitled *Building Systems from Commercial Components* [Wallnau 01]. BSCC takes the point of view of an architect and engineer, and defines a process based on risk analysis, realization of model problem, and repair of residual risk. Like APCS, this process specifies the necessity of using an iterative or spiral approach. This work is reflected in APCS through the stress on prototyping, and the primacy of risk reduction in the development process.

The third project, and the one most intimately connected with APCS, was the parallel development of an instantiation of APCS tailored to the dimensions of the Rational Unified Process® [Kruchten 00]. This process, known as the Evolutionary Process for Integrating COTS-based systems℠ (EPIC℠), is thoroughly documented [Albert 02]. The conceptual bases of both EPIC and APCS stemmed from the same research effort.

6 An SEI technical report on product evaluation is to be published.

® Rational Unified Process is a registered trademark of Rational Software Corporation in the United States and in other countries.

℠ The Evolutionary Process for Integrating COTS-based Systems and EPIC are service marks of Carnegie Mellon University.

# 3 The APCS Process

## 3.1 Guiding Principles

To guide the APCS process definition, we articulate six high-level principles, all of which are derived from the fundamental realities of the independence of the COTS software marketplace and the impact of COTS products on requirements. We present each principle by pairing a condition (i.e., a *de facto* truth of COTS-based projects) with the necessary practice that the condition compels. Each principle is in some way the driver of one or more of the details of the APCS process.

1. The requirements of a COTS product[7] are independent of the requirements of a system that uses the product. As a result, the place where flexibility is needed is in the system's requirements, not in the products that implement that system. Hence,
   *The requirements process must celebrate flexibility: requirement definition must be delayed and negotiated, and the requirements minimized.*
2. Any COTS product naturally imposes its own constraints on the end-user processes it automates. Hence,
   *Use of any COTS product necessitates documenting, and probably re-engineering, of existing end-user processes.*
3. Marketplace change is ongoing and compelling, with continuous technical opportunity for growth. Hence,
   *A COTS-based system must be continually re-formed and evolved throughout its lifetime.*
4. Imperfect knowledge of COTS product features is usually all that can be achieved. Hence,
   *Hands-on product evaluations are mandatory; these must be budgeted, scheduled, and accepted as a fundamental and frequent project activity, as central as requirements and design.*
5. Different COTS products are built separately, and in most cases are not designed to work together. Hence,
   *Prototyping of the integrated collection of COTS products, from the earliest possible moment and throughout the system's life, is a basic necessity.*
6. The use of COTS products is predicated on a set of binding contractual commitments, encompassing business, financial, and technical issues; once made, these commitments have a lasting and pervasive effect. Hence,
   *Satisfactory contractual commitments can only result from the active participation of end users, testers, and other stakeholders, continuously throughout the entire period from project inception through sustainment and maintenance.*

[7] That is, the requirements to which the product was developed, resulting in the features that it has.

## 3.2 Process Overview

At the highest level, the APCS is a spiral process consisting of a series of iterations, each of which has a similar shape and employs similar activities. An iteration begins with planning, then moves to a period of discovery, and then assembles an executable prototype of the system, which is then assessed as part of the preparation for the next iteration. It is implicit that the duration of an iteration will generally be brief, along the lines of the common understanding of "rapid prototyping." Iterations continue until one or more systems have been constructed that are deemed acceptable for deployment. Given the realities of the COTS marketplace, further iterations will be necessary throughout the system's sustainment and evolution.

Each iteration is composed of a set of activities that often overlap and are often interdependent. We divide these iterative activities into three kinds:

- *discovery* activities that gather and refine knowledge about the system
- *assembly* activities that construct a prototype
- *assessment* activities that determine the success of the iteration and plan the next

Before describing these activities in greater detail, we first must put them in the larger context of all of the activities involved in software development.

### 3.2.1 Other Classes of Activities

We suggest that the iterative activities comprise only one of three classes of activities that are operative in system development:

- *iterative* activities, which are typically short term and oriented toward engineering
- *pervasive* activities, which tend to be long term and are organizational in scope
- *executive* activities, which tend to be event driven and deal with decision-making

Note that these categories are not mutually exclusive. For instance, many of the activities of management, which we label either executive or pervasive activities, are themselves iterative, and many kinds of decisions are needed within many engineering activities regardless of their identity as iterative or otherwise.

**Pervasive** activities concern organizational capability and project-specific management activities. Pervasive activities do not depend on the spiral iterations, but are continuous, without clear stopping or starting points. For instance, one rarely says "There—my configuration management is finally done for this iteration...." Pervasive activities have sometimes been described as "technical management," since they share aspects of both the technical and managerial domains [PSESWG 93]. Unlike iterative activities, pervasive activities also tend to be independent from each other. The following are common pervasive activities:

- configuration management
- license management
- vendor relationship management
- contract tracking and oversight

**Executive** activities tend to fall purely into the managerial domain. They include such management-centric tasks as scheduling and costing, as well as making decisions in the technical arena. Executive activities also include most procurement-specific activities (e.g., for a government acquisition). Although some executive activities are relatively independent of the spiral iterations, most are closely bound to the iterations, since the decision points of many iterative activities are precisely where executive decision activities must come into play. Executive activities include:

- cost estimating
- contract negotiation
- project oversight
- system deployment

The pervasive and executive activities are as significant to system development as the iterative, but will not be described in the present paper. With the exception of a large-scale activity, which we label "Manage Project," we will not attempt to incorporate them into APCS, whose proper province is the collection of activities whose essential focus is creating a system using COTS products.

We now concentrate on describing the three kinds of iterative activities: Discovery, Assembly, and Assessment.

## 3.2.2 Discovery

The Discovery activities consist of examining all of the pertinent sources of knowledge, both to *gather* and to *refine* that knowledge. It is unavoidable that gathering and refining proceed simultaneously. It is also unavoidable that they will have a certain chaotic quality.

*Gathering* takes many forms, for example, consultation with users and other stakeholders (which reflects the familiar task of requirement elicitation); articulation of business processes; and especially, evaluation of COTS products. These different "gathers" are themselves independent. Thus, evaluating the features of candidate COTS products and eliciting desirable features from end-users are separate tasks, at least conceptually, and it is seldom possible to assert that one should necessarily precede the other. Similarly, understanding the constraints of legacy systems can be simultaneous with examination of business processes (and, most likely, planning their re-engineering). Throughout all of the gathering activities, some items of information will invariably contradict some other items of information, since they derive from conflicting needs of different stakeholders.

The analysis and harmonization of these conflicting needs, reflected in these different pieces of knowledge, occurs through what we term the *refine* activity. Here, the different kinds of knowledge are analyzed and the technical definition of the emerging system is evolved. Refinement will find gaps in the knowledge (e.g., evaluation of a COTS product may not have produced enough information about its hardware resource needs), and will also find conflicts (e.g., the necessary interfaces with the legacy systems may be incompatible with any of the available products).

In the case of a gap, the solution is straightforward: further knowledge must be gathered. The presence of a conflict, however, implies that negotiation must occur between different stakeholders. Conflicts can entail a very difficult kind of negotiation, since the pieces of knowledge (and hence the decision factors involved) will often be of very different classes. For instance, the end-users may strongly prefer a product that is quite expensive, but whose workings are very compatible with the organization's business process. A competing product may be less desirable from the end-user point of view, but may also be far more affordable. A decision must be made between ease-of-use and budget, for which no method is readily available. The need for this type of comparison (often called "apples and oranges") is a common occurrence in COTS-based development, and it is during the gathering and refining of knowledge that such conflicts are identified.

Because this is so time-consuming and expensive, it is also another justification for our insistence on delaying firm requirements definition as long as possible. Only after considerable discovery has occurred and considerable tradeoffs among competing stakeholders have been made is it reasonable to define any but the most compelling and absolute conditions that the system must satisfy.

There will be constant (and non-uniform) oscillation between the *gather* and *refine* activities: refinement leads to further gathering of information, which must then be refined, and so forth. Discovery thus cycles back and forth, with different activities simultaneously gathering and refining various kinds of knowledge. The relationships between the Discovery activities are illustrated in Figure 4.

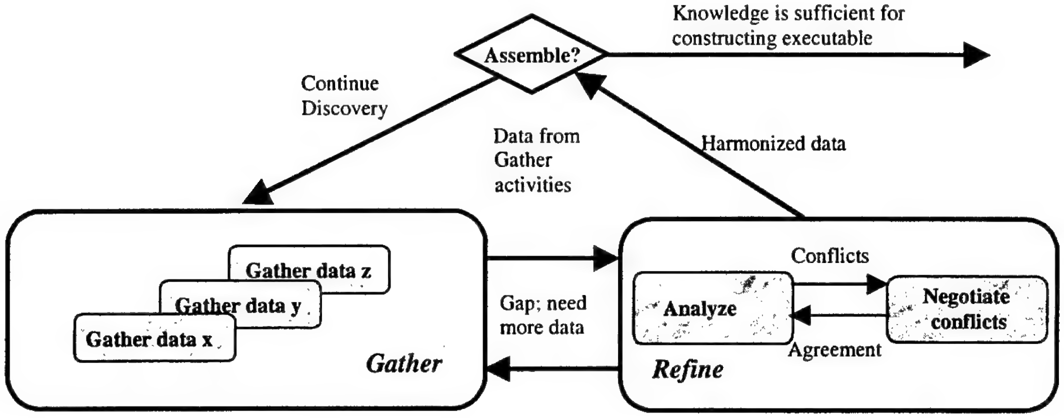


*Figure 4: Discovery Activities*

### 3.2.3 Assembly

We use the word "assembly" for the activities that construct the prototype. These activities will tend to reflect a traditional sequential process, in which requirements precede detailed design, then coding, integration, and testing. But while the activities themselves appear traditional, their relative proportions may be quite different. In particular, the presence of COTS products implies that a significant element of system construction, and perhaps most of it, will involve integration of one or more COTS products, in contrast to coding in the traditional sense. Hence, comparatively little time may be spent coding, in contrast to traditional or custom development. Note, however, that even with a large COTS presence, there will usually also be at least some code to write, since a system of any complexity will likely require some sort of glue code or custom functionality.[8]

Assembly begins when Discovery stops. While there is no objective guide to indicate when the knowledge gained by the Discovery activities is sufficient to create an executable prototype, we can list some indications. Most important is that the process specifies the existence of a set of Iteration Objectives and a Detailed Iteration Plan. (See Section 4.2.) These will jointly define the intended nature of the prototype and will make it possible to gauge the scope of the anticipated knowledge that will be needed. In addition, the Discovery activities may have generated questions and issues whose immediacy can only be answered by an executable system. And finally, agreements among stakeholders may have reached a point that validation (i.e., through execution of the prototype system) is an obvious and desirable point of closure. There may well be other factors that influence the decision to assemble, such as budget constraints or users' demands for an immediate solution.

[8] Note also that if modification to COTS products is involved, this may require a very considerable amount of coding, and may well be a highly difficult task.

In APCS, in place of a completed set of system requirements, the previous Discovery phase has produced a collection of goals, expectations, hypotheses, and desired behaviors that we hope to derive from the prototype. These are treated, *for the duration of this iteration*, as requirements, though local in effect. Viewed from a global vantage point (i.e., this iteration in the context of many iterations), these "requirements" are in effect a set of hypotheses that will be proven true or false by the success of the prototype. Such "candidate requirements" might include decisions to try particular Graphical User Interfaces (GIUs) with end-users, trials about how several COTS products can successfully interface, scalability or throughput considerations for one or more COTS products, and so forth. After the prototype is completed and assessed, some (or all) of these local requirements may be accepted as genuine requirements, reconsidered, revised, or rejected (see Section 4.2.3).

Even in the first iteration, we consider that Assembly should produce a prototype that is, in some manner, an executable version of the full system. One reason for this stipulation is to ensure that each prototype can be inspected and assessed not only by a small subset of stakeholders, but by the widest possible community. Thus, even though a particular prototype may have a primary objective of demonstrating the overall business process to end-users, that same full system prototype will very likely satisfy objectives that have at least some relevance to other stakeholders (e.g., COTS product performance data, budget information, etc.)

That said, it is still true that the early prototypes will undoubtedly have considerably less than complete functionality; a few, and perhaps many, of the functions will be stubs. But however incomplete, we look for a "full system prototype" to exhibit such features as the following: the system exhibits end-to-end functionality; it automates end-user operations and can be assessed by end-users; the features that are implemented do not preclude satisfying any remaining requirements.

One final note on Assembly: We fully expect that other forms of prototyping—often called "prototyping in the small" and performed via localized test cases, extensive use of model problems as described in BSCC, and so forth—will be part of the process. These may be part of many Discovery activities, and may also occur in many different ways throughout the Assembly activities as well. Whatever role these forms of prototyping may play, however, we *do* stipulate that each iteration must focus on and work toward a stand-alone executable version of the system, not a paper version or a specification or a small portion of the system. It is this full-system executable that we refer to as "the prototype."

### 3.2.4 Evaluation and Assessment

This activity occurs at two distinct, though overlapping, levels. At one level, that of the prototype only in its own context, the prototype must be evaluated to measure actual outcome compared with the expected outcome. This evaluation will consider the degree to which the prototype satisfied its "local requirements."

At a quite different level, the iteration itself must be assessed. It must be determined, for instance, whether the objectives chosen were met, whether they were appropriate, or whether the results of the iteration indicate any shift in schedule or budget. The results of the iteration are also the basis for planning the next iteration.

## 3.3 Ongoing Iterations

The above description defines a single iteration, but a real project will comprise several iterations before a system is fielded. Each iteration has a similar shape: Discovery, Assembly, Assessment (Figure 5). The differences between iterations lie in (1) the nature and scope of knowledge that must be gained; (2) the expected reduction in risk that is achieved; and (3) the increasing capability of the prototypes to be deployed.

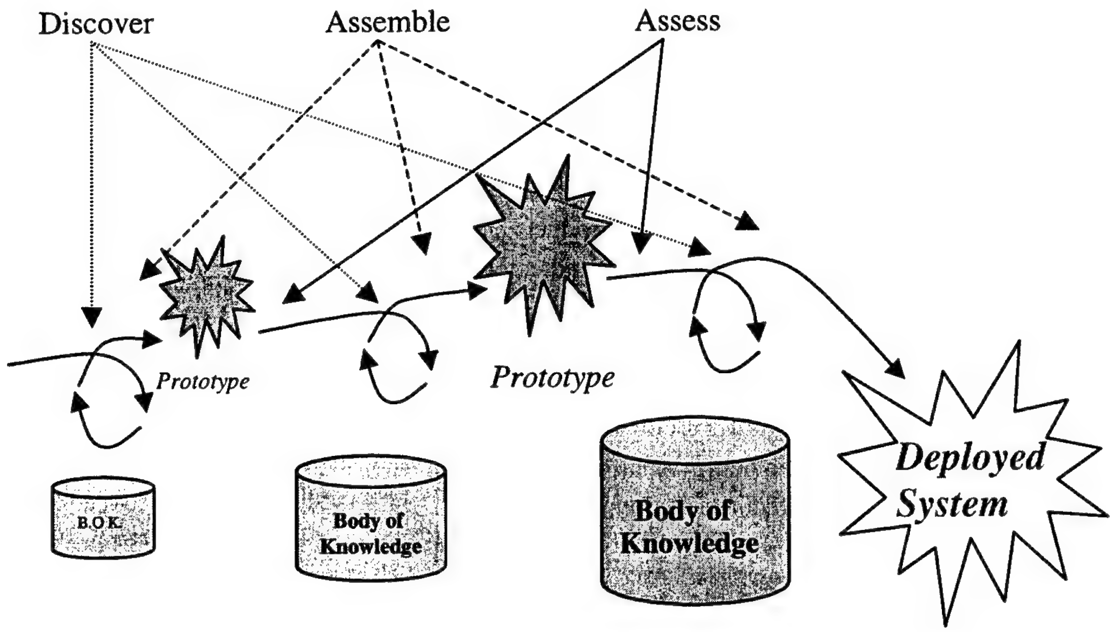


*Figure 5: Ongoing Iterations of APCS*

Figure 5 depicts iterations that progress serially. However, it is very possible that there will be a number of iterations that progress simultaneously. This may be inevitable for a program of any complexity. Hence, as an iteration begins assembling the prototype, a new iteration, commencing with a new period of Discovery, might begin. This has the obvious management advantage of balancing work distribution among analysts, programmers, testers, and so forth. It also permits a much faster incorporation of stakeholder responses into ongoing system releases.

## 3.4 System Deployment

It is clear that the decision to deploy a system is related to meeting the system's requirements. Indeed, in a traditional development paradigm, the deployment decision would be made after all of the requirements had been met and the system accepted by the acquirer. Of course, there were always exceptions to the rule. For example, in times of war, where deployment of a partially constructed system could still provide benefit to the users, a decision for early deployment might be made. However, in a COTS-based, iterative process, where every iteration results in a prototype that is an approximation of the desired system, there are many more opportunities to make the deployment decision. Indeed, the assessment of each prototype should involve determining the suitability of the prototype for deployment. And just as there is no objective guide for deciding if knowledge is sufficient to assemble a prototype, there is also no objective guide for determining whether a prototype is suitable for deployment. We should be able to rely on familiar decision criteria, since the deployment decisions for COTS-based systems are not that different than those for traditional systems, and the criteria and judgments that suffice for a custom system retain their validity for a COTS-based system

It is the case, however, that the presence of COTS products adds a new element to the deployment decision. Such factors as imminent product release, a vendor's announcement of price changes for licenses, or the appearance of new products and technologies may suggest either delaying a system's deployment or hastening it. An indicator for deployment might be that needed end-user feedback for subsequent evolutions, using live data and full operational conditions, can only be gained through a deployment to the field. Perhaps more significant is that, for a COTS-based project (and for any project that uses a spiral development approach), the contractual mechanisms will likely be different than for a custom system. There may be some accommodation of, or even a preference for, a phased deployment, with fielding of partial capability. This may have contractual implications that affect the deployment decision.

Finally, the very notion of deployment as a "big bang" event is altered in COTS-based development. Any system that makes heavy use of COTS products will necessarily be replacing and upgrading its key components in an ongoing manner, as the vendors release new versions of their products; these replacements will have begun even before any deployment occurs. And given the context of ongoing upgrades to COTS products coupled with several early deployments of partial capability (which is now favored by government directives), it is often difficult to distinguish **any** single event as "the" deployment of a system.

# 4 A Model of APCS

We model APCS using Integrated Definition functional modeling (IDEF0), a method that has been incorporated into several DoD recommendations on standards and modeling. IDEF0 is itself a modification of the Structured Design and Analysis Technique (SADT) described by Marca [Marca 88]. In IDEF0, boxes depict activities, and are hence labeled with verbs. The arrows that link the boxes indicate either inputs (left-side arrows), outputs (right-side arrows), controls (top-side arrows), or mechanisms (bottom-side arrows). In the diagrams in this paper, we have omitted all bottom-side arrows (i.e., indications of the mechanisms for implementing an activity).

Seen at the highest level, APCS is similar to almost any overall description of a spiral process. As shown in Figure 6, there are three main steps. One is a governing management activity that guides and directs the program execution. The second step is the performance of an iteration. In the third step, the iteration is assessed. This provides feedback to the management activity, where plans are revised and updated as needed, and the process cycles through several iterations until one or more systems are fielded.

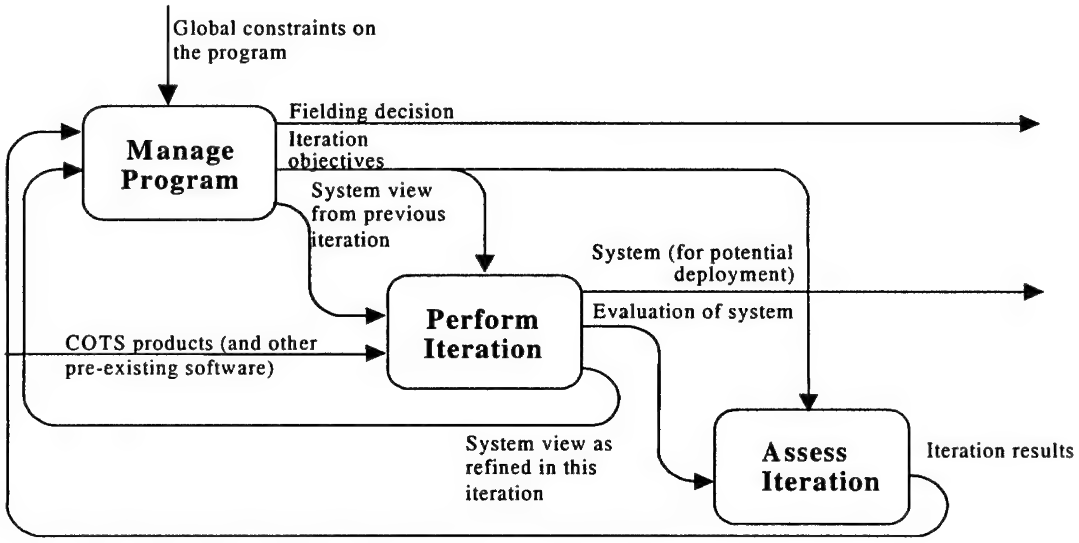

Global constraints on the program
Manage Program
Fielding decision
Iteration objectives
System view from previous iteration
Perform Iteration
System (for potential deployment)
Evaluation of system
COTS products (and other pre-existing software)
System view as refined in this iteration
Assess Iteration
Iteration results

*Figure 6: Top-Level View of APCS*

## 4.1 Manage Program and Assess Iteration

Of these three high-level steps, we shall describe *Manage Program* and *Assess Iteration* only briefly here, since as noted earlier, the details of both of these are largely concerned with pervasive and executive activities. They may be described in a succeeding report that considers how contracting, budgeting, and so forth are affected by the presence of COTS software. Note, however, that many of the executive activities of *Manage Program* are simultaneous with and closely related to, the iterative activities that comprise *Perform Iteration.*

One critical output of *Manage Program* is the set of Iteration Objectives that are defined for each iteration. These include the technical and programmatic items that the iteration must accomplish, and the constraints (e.g., schedule, available resources) that the iteration must satisfy. A second output is the evolving System View, which is modified and refined by the ongoing spirals of *Perform Iteration.* Since the end product of each iteration is a prototype system, these prototypes become progressively more viable candidates for deployment, and at some point, a decision must be made to field the system (or, in the case of repeated failure, to terminate the program). This fielding decision is the third output from the *Manage Program* step.

*Assess Iteration* is governed by the objectives defined for the iteration; these have been produced by *Manage Program.* The only input to this process step is the evaluation of the prototype system that is produced during *Perform Iteration.* There is one output, namely, an assessment of the iteration: how it met (and did not meet) its objectives, the risks it considered and whether they were mitigated in a satisfactory way; and other issues that relate to the overall program's goals.

The remainder of this paper will concentrate on the middle step, *Perform Iteration.* We will break it into its constituent substeps, and in some cases, continue decomposition to another sub-level. The end result of this breakdown will be a set of "leaf" substeps. Purely as a simplifying device, we start our consideration of *Perform Iteration* as though it were itself the top-level node, a decision that considerably simplifies the IDEF0 version of the model, particularly the nomenclature of the substeps.

## 4.2 Perform Iteration

We depict *Perform Iteration* as a single process step in Figure 7.

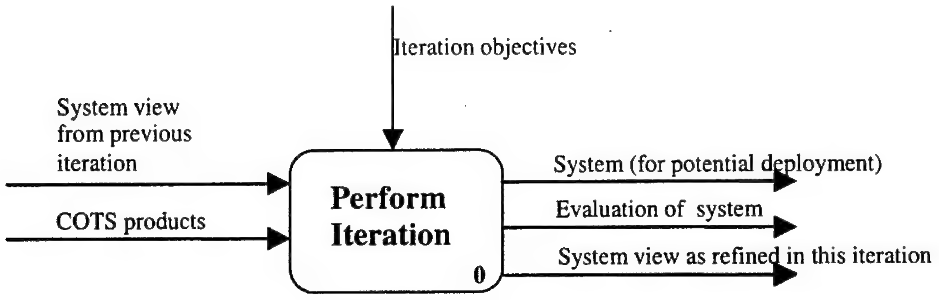

Iteration objectives
System view from previous iteration
COTS products
Perform Iteration
0
System (for potential deployment)
Evaluation of system
System view as refined in this iteration

*Figure 7: Perform Iteration*

*Perform Iteration* is controlled by the Iteration Objectives that have been defined for the iteration by *Manage Program*. One input to this process step is the System View as defined by previous iterations. If this is the first iteration, then this is, in effect, a default input, consisting of whatever is known about the system at project initiation. This might include such broad-based artifacts as a Statement of Objectives, a Mission Needs Statement, or similar high-level documents. If one or more iterations have occurred, the System View will include the requirements as they have evolved so far; the evolving architecture and design of the system; documentation about products evaluated and selected; statements of stakeholder needs and preferences, and so forth. The other input to *Perform Iteration* consists of the COTS products that will be used in the prototype system.

There are three outputs: an executable system—the prototype; an evaluation of that system against its local requirements; and an updated collection of the artifacts that comprise the System View. As iterations progress the executable increasingly becomes a candidate for fielding, and the System View becomes more and more complete.

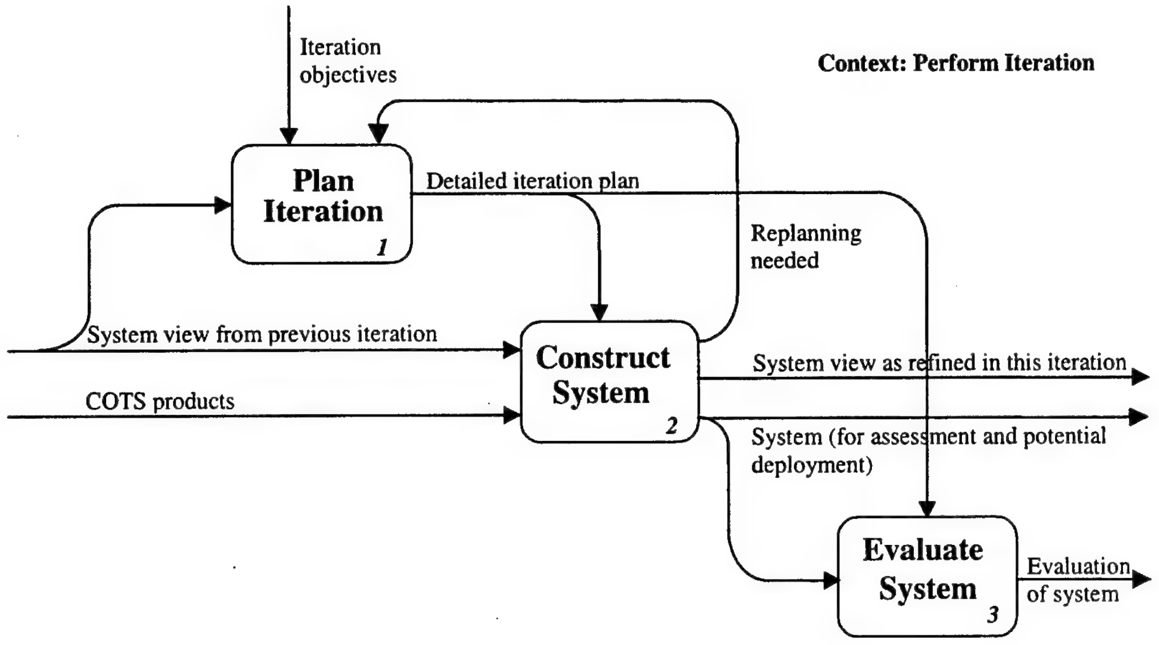

Iteration objectives
Context: Perform Iteration
Plan Iteration
1
Detailed iteration plan
Replanning needed
System view from previous iteration
COTS products
Construct System
2
System view as refined in this iteration
System (for assessment and potential deployment)
Evaluate System
3
Evaluation of system

*Figure 8: Detail of Perform Iteration*

Figure 8 shows that the *Perform Iteration* process step logically breaks down into three substeps: *Plan Iteration; Construct System; Evaluate System.* The first substep produces a Detailed Iteration Plan that governs all of the remaining steps and activities of the iteration. The second substep produces the executable version of the system as well as all of the updates and transformations of the evolving System View. The third substep produces an evaluation of the prototype, that is, the degree to which it met the specific goals and objectives—the local requirements—for the current iteration.

Note that the Iteration Objectives that **control** *Plan Iteration* reflect a global perspective: these are formulated at the *Manage Program* level. By contrast, the Detailed Iteration Plan that is the **output** of *Plan Iteration* is local and defines how the iteration will unfold purely on its own terms. This iteration plan is also the controlling agent for almost every other activity carried out during the iteration. We will discuss this distinction between global and local perspectives at greater length in the following section.

One further note about Figure 8 is the feedback loop from *Construct System* back to *Plan Iteration.* Any plan is likely to require modification, perhaps often, and this feedback loop is the logical mechanism that describes how replanning relates to the other activities that take place during an iteration.

### 4.2.1 Step 1: Plan Iteration

Although we have asserted that a COTS-based approach has significant differences from custom software development, it is nonetheless true that many things are largely unchanged. Of these, the planning process is perhaps the area that is least changed. To be sure, the *scope* of things that planners must account for is more diverse, but the actual planning process itself, particularly in its scheduling and budgeting aspects, has many parallels to the traditional process. We are therefore most interested in those elements of planning that may be unfamiliar, whether because of COTS or because of performing a spiral process, and will focus on those in the description below.

In any spiral process, it is vital that those who execute it understand the necessary division of perspective between the overall project level and the level of the individual iterations. That is, a planner who has the entire project in perspective will have determined which useful subset of the overall project's objectives should become the objectives for a particular iteration. This kind of planner is, in effect, juggling many things at the same time, and has (or should have) alternatives, fallbacks, and other contingencies in mind at all times. This planner's global perspective also requires an explicit notion of the goals for risk reduction and mitigation for each iteration.

By contrast, a planner whose responsibility is an individual iteration has a more concrete task. This person receives a set of iteration objectives. His or her role is to define a plan that will accomplish those objectives, and which includes such constraints as performing the iteration in a specified time and using specific resources. What this person must **not** do is confuse issues, constraints, and goals of the overall project with those of the individual iteration.

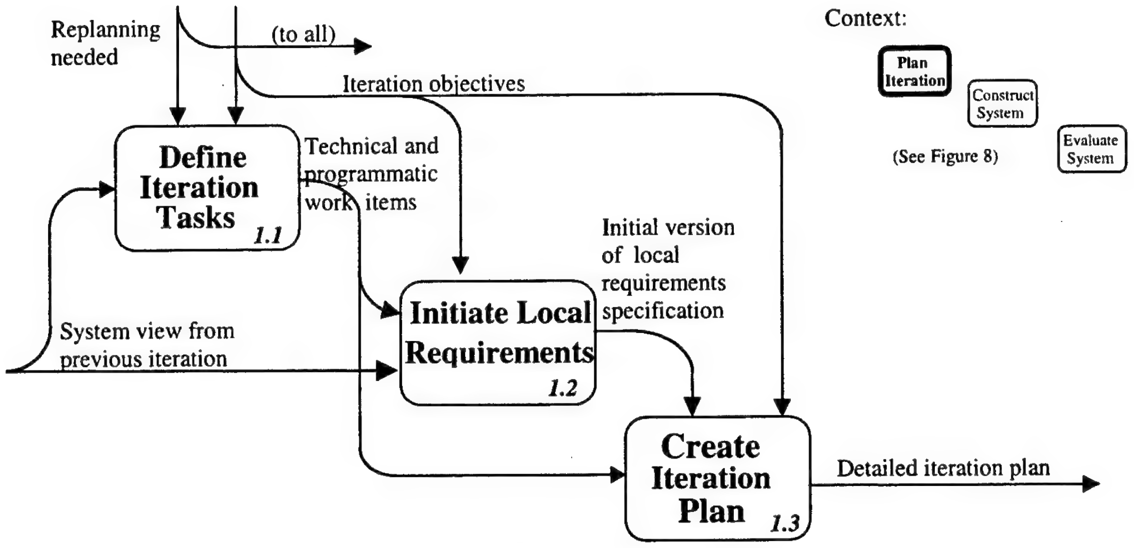

Replanning needed
(to all)
Iteration objectives
Context:
Plan Iteration
Construct System
Evaluate System
(See Figure 8)
Define Iteration Tasks 1.1
Technical and programmatic work items
Initial version of local requirements specification
Initiate Local Requirements 1.2
System view from previous iteration
Create Iteration Plan 1.3
Detailed iteration plan

*Figure 9: Further Decomposition of Plan Iteration*

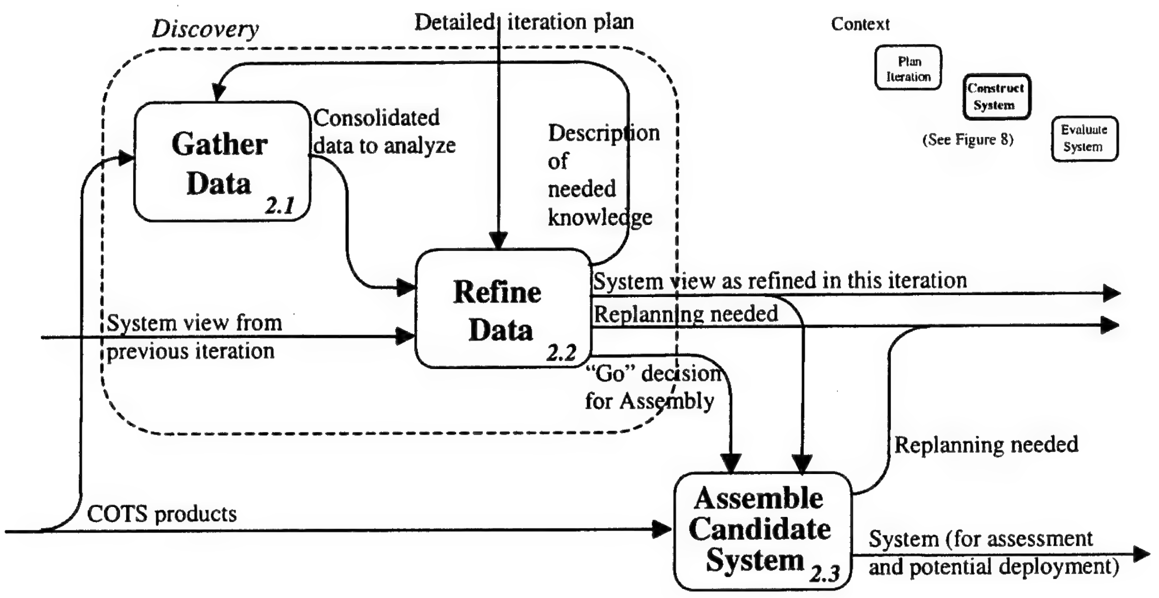
Figure 9 shows that there are three substeps in *Plan Iteration*. The first defines the primary work of the iteration: the technical work items to be accomplished and the functional properties that will be prototyped. These work items may have an imperative quality ('integrate x with y"), or may have an inquisitive quality ("determine whether end-users can accept z"). It is likely that in an early iteration, most of the work items will have an inquisitive character, since early iterations will generally be unearthing unknowns. A preponderance of imperative work items suggests that many of the decisions about the system have been made, so the work items will reflect choices already made.

The second substep, *Initiate Local Requirements,* produces the first version of local requirements that govern the iteration; these are derived from the iteration objectives. APCS is based on the concept that the requirements process must be flexible, and when creating these purely local requirements we can see this concept in practice. The local requirements that emerge from *Plan Iteration* are themselves only a first version, to be refined through gathering, refining and negotiating among stakeholders. Only just before Assembly (in *Form System View*, Step 2.2.3 in Figure 11) will even these local requirements be finally solidified. [Note also that the degree to which these local requirements will become actual requirements will depend on how this (and subsequent) iterations unfold.]

Finally, *Create Iteration Plan* is a standard planning step that defines milestones, workloads, and other familiar planning issues. This information is documented in a formal Detailed Iteration Plan, which guides the remainder of the work of the current iteration.

### 4.2.2 Step 2: Construct System

Figure 10 shows that there are three substeps in *Construct System*: *Gather Data, Refine Data,* and *Assemble Candidate System.* The first two substeps perform the work of Discovery (see Section 3.2.2). As noted earlier, these two substeps have a necessarily chaotic character, and may consume the largest segment—at least in terms of time, and perhaps even of resources—of the Construct System step.



Discovery
Detailed iteration plan
Context
Plan Iteration
Construct System
(See Figure 8)
Evaluate System
Gather Data 2.1
Consolidated data to analyze
Description of needed knowledge
Refine Data 2.2
System view as refined in this iteration
Replanning needed
System view from previous iteration
"Go" decision for Assembly
Replanning needed
COTS products
Assemble Candidate System 2.3
System (for assessment and potential deployment)

*Figure 10: Further Decomposition of Construct System*

*Gather Data* consists of acquiring raw information: eliciting stakeholders' needs, evaluating products, uncovering facts about legacy systems, and other such activities that build up the Body of Knowledge. *Refine Data* transforms this raw data into the evolving System View. *Refine Data* also controls the various Gather activities by describing the specific knowledge that must be gathered. These two substeps (*Gather* and *Refine*) will rarely be independent of each other, and will almost always be simultaneous. If they meet any insurmountable barriers, it may be that the iteration plan will need to be amended; it may even be the case that the goals of the iteration will need to be reconsidered. *Assemble Candidate System* resembles a traditional system development, though the relative proportions of coding and integration may be quite different from those of a traditional effort.

#### 4.2.2.1 Gather Data

All of the "gather" activities unearth knowledge; the specific nature of each activity is governed by the type of data to be gathered. Four of the likely types of knowledge are

- stakeholder inputs
- COTS product data

- business process data
- external system data

The "gather" activities are independent from each other, with no particular relationships necessary between them, but the boundaries between the types of data are fluid. For instance, it is often difficult to distinguish some stakeholders' needs and business process issues; the "business process" matter of interfacing with a legacy system may also be considered a stakeholder concern. Any redundancy that occurs between the different "gathers" will subsequently be resolved through refinement. Note that the proportions of the "gathers" may vary, depending on the nature of the particular system and its context; the relative proportions will also vary depending on whether an iteration is early or late in the process.

Once the raw data has been gathered, it must be formatted and consolidated before it is refined. The essential nature of this consolidation is the same, regardless of the type of data.

#### 4.2.2.2 Refine Data

This process step is unavoidably complex, and requires further decomposition, which is shown in Figure 11. This step is perhaps the most difficult aspect of APCS, and is the crux of the distinction between COTS-based and custom software development. It entails comparing stakeholder needs, the constraints of legacy systems, the business process, and candidate COTS products, and then transforming these conflicting elements into a coherent System View that is the basis of assembling an executable system.

All substeps in *Refine Data* are governed by the Detailed Iteration Plan. There are two external inputs: the data from the various Gather activities, and the current state of the System View. The initial substep, *Analyze Data*, further refines negotiable and non-negotiable items into the prototype's requirements. A significant task of this analysis is also to prioritize the negotiable requirements as much as possible. The analysis will reveal conflicts, which are considered in *Negotiate Conflicts*. *Analyze Data* and *Negotiate Conflicts* will both also uncover gaps in the available data; *Determine Data to Be Gathered* is the transformation of these gaps (and any other needed knowledge) into some form of specific instructions to the various Gather activities. (Note that for the first iteration, *Determine Data to Be Gathered* may be the only activity that is performed here, since little knowledge will initially be available for refinement.)

*Form System View* is actually a complex of activities whose extent depends on the artifacts that comprise the System View. Input to this substep is the existing System View together with the evolving and growing set of agreements on requirements and preferences for this iteration. Any resolutions of conflicts are also, in effect, agreements, so these are also inputs to this substep. One major work of *Form System View* is to finalize the set of local requirements that were defined initially (during *Plan Iteration*) and that have undergone refinement as the iteration has progressed through ongoing analysis and negotiation.

Note also that as the System View is evolving, the ongoing changes to its state are necessarily fed back to *Analyze Data.* This is because analysis depends on knowing the present state (e.g., all agreements that are building up during this iteration), not merely the System View of the previous iteration.

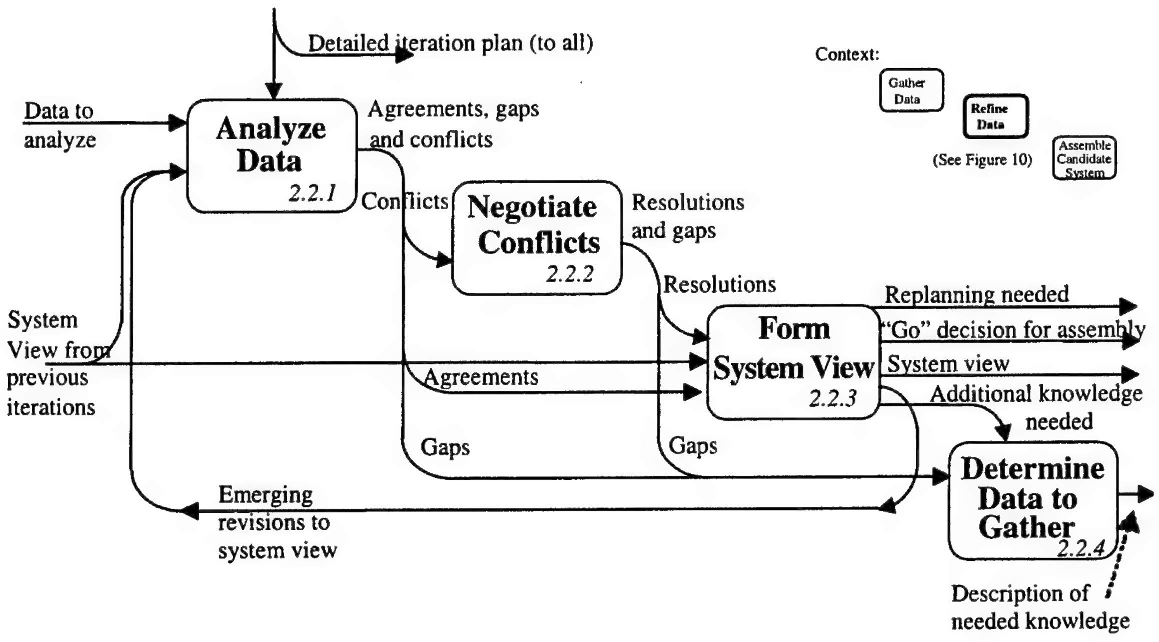

Detailed iteration plan (to all)
Context:
Gather Data
Refine Data
Assemble Candidate System
(See Figure 10)
Data to analyze
Analyze Data 2.2.1
Agreements, gaps and conflicts
Conflicts
Negotiate Conflicts 2.2.2
Resolutions and gaps
Resolutions
System View from previous iterations
Agreements
Form System View 2.2.3
Replanning needed
"Go" decision for assembly
System view
Additional knowledge needed
Gaps
Gaps
Determine Data to Gather 2.2.4
Emerging revisions to system view
Description of needed knowledge

*Figure 11: Low-Level Activities of Refine Data*

In addition to the transformed set of artifacts, *Form System View* also produces two other outputs. The first occurs only when an irresolvable issue arises that prevents a coherent System View from being formed. This may be a technical matter, or may be purely programmatic (e.g., the only acceptable COTS product costs far beyond the available resources). In such a case, some replanning is necessary, either at the local level of the present iteration, or at the overall program level.

The other additional output of this substep occurs when no roadblock occurs, and a coherent System View, sufficient to assemble an executable, has been achieved. This takes the form of a decision to begin the activities of *Assemble Candidate System.*

### 4.2.2.3 Assemble Candidate System

This process step is straightforward, and requires no further decomposition. In fact, in assembling a prototype, even with the presence of COTS products, it is likely that many familiar parts of the custom development process—detailed design, integration, system test—will be relatively unchanged. Even in the presence of less familiar activities, such as modifying COTS products, or debugging without access to source code, the essential process of assembly still bears a strong resemblance to a traditional system construction process. There is therefore little need for extensive description of the assembly process step; whatever processes an organization has used in the past to develop software will have a strong influence on how this part of *Perform Iteration* is conducted. The typical activities will include

- Create detailed design.
- Perform needed modifications.
- Write custom code.
- Integrate components.
- Test.

We will, however, call attention to a few of the changed circumstances that may derive from the use of COTS products.

Creating a detailed design may be difficult to separate from product evaluation, solving model problems, and other forms of Discovery. It is therefore likely that the designers will begin their work with a considerable amount of design (and even very detailed design) already accomplished.

It may be that the design requires that one or more COTS products be modified,[9] a task that is often quite difficult. Even when using the vendor's mechanisms (as, for instance, in many Enterprise Resource Planning systems), the modification effort can be considerably more complex and time-consuming than originally planned. We therefore caution that modification should be undertaken only when absolutely necessary, and in awareness of all its ramifications.

Unless the system is entirely "turnkey," it will be necessary to write some custom code. When several heterogeneous COTS products are elements in a system, a considerable amount of code may be required. This code—whether "glue," middleware, or an entire component of significant functionality—may be trivial or complex. It may span a spectrum from simple scripts to extensive use of such capabilities such as common object request broker architecture (CORBA), Java Beans, and other comparable technologies. Complex information systems in particular may require writing the necessary code for custom-made screens, customization of business rules, and schema definition.

We have already noted that the relative proportions of coding and integration will differ from traditional construction. Integration will also occur earlier, and some integration may even precede any coding. This is not the familiar case, since in custom development, the parts must exist before they are put together. But some, perhaps many, COTS parts will be on hand at the very beginning of the assembly process, and integration may well be the first activity that is performed.

---

9 We will not here pursue a discussion of the subtle distinctions commonly made between "tailoring," "customizing," and "modifying." See Carney for an extended treatment of these topics [Carney 01]. In the above discussion, "modification" refers to changes of any sort that the system builder makes to a COTS product.

Finally, heterogeneous COTS products (regardless of how one expects them to interact) often behave in unexpected ways and require a notion of debugging different from the traditional norm. In fact, the term "debugging" is itself a misnomer, because the products individually may be behaving properly, but their interconnection is not. The traditional tools of debugging, particularly examination of source code, are not applicable here, and a set of other techniques are needed. See Hissam for a description of some common methods that can assist in this process [Hissam 98].

### 4.2.3 Step 3: Evaluate System

The system evaluation step will take place in the context of a single iteration, even though it may have implications for the overall project. This step is critical to the overall success of the project, as well as this particular iteration, because it provides the foundation for continuing on to the next spiral/iteration.

As with our description of *Plan Iteration*, we note that evaluating the prototype system has many resemblances to the system evaluation process used for a custom system. Certainly the presence of COTS products presents some challenges not ordinarily encountered, but there is little essential change to the testing and evaluation process itself. As before, in describing this step we will focus on those elements that may be unfamiliar or different because of the presence of COTS products or the use of a spiral process.

Evaluation of the prototype system is done in the context of the local requirements for this iteration; we seek to find how well these local requirements have been satisfied. That is, it is a *system* test (as opposed to unit or integration test), but its province is only the requirements that were determined in the *Determine Local Requirements* step of ***Plan Iteration*** to be relevant to this iteration.

Some part of evaluation at this level has a relationship to quality assurance. With custom-developed systems, the focus of quality assurance has usually been on assuring that the software is defect-free. This is problematic for a COTS-based system. First, we know that it is highly unlikely that we will be getting defect-free products from our COTS vendors. In addition, we will not have the source code available to us for practicing many of the techniques that quality assurance has relied on in the past. Ultimately, we find instead of using the concept of "defect-free," we might postulate some other indications of "quality." For example, the notion of "user satisfaction" seems related to quality, but it is hard to measure; there is some sense that it meets expectations—which may extend beyond the requirements that have been written down. Measurable things such as the classical measures of hardware quality—mean time between failure and mean time to repair—seem less appropriate for software, but could be explored further in this context.

# 5 Summary and Conclusion

In this paper, we have described a generic process for COTS-based system development. Earlier, we enumerated six guiding principles. At this point, having described the process in detail, we can suggest the following salient features as a summary:

- Requirements stand in a different relation to a COTS-based system than they do in relation to a custom-built system.
- Small-scale steps, delayed decisions, and constant negotiation and trade-offs among competing constraints are the watchwords for most APCS process activities.
- Minimizing and buying down risk is the foundation of the APCS spiral process.
- Traditional distinctions are blurred in a COTS-based process; system design overlaps with product evaluation, and development and maintenance sometimes become indistinguishable.

To conclude the description of APCS, we briefly discuss the issue of instantiation of the process. We also indicate the further research that is needed to complete a full description of a COTS-based system development process.

## 5.1 Instantiation

The APCS process as we have described it in this paper will require instantiation, and we have already referred to one such instantiation [Albert 02].

One element of instantiation is intimately related to the system to be built and the project to be carried out. For instance, if the project is very large, and will be executed over a period of years, the instantiated process will need to be formal and well documented. If the project is to be done by a small number of people and will consume no more than a few months, then such a heavy-weight process would be a burden, and would likely constrain rather than facilitate success.

Regardless of project size, however, the act of instantiating APCS will involve definition of the specific artifacts that must be created as part of the process. Even for a very small-scale process, the outputs of any step—for instance, a specification, or any such artifact that influences or constrains succeeding process steps—must be well understood by the process participants, and hence must be explicitly defined before the process can unfold.

There are several things that affect both the identity and the content of such artifacts. For instance, if instantiating APCS in the context of government regulations, there are numerous documents required by regulation that will necessarily become elements of the instantiated process. In the EPIC instantiation, many of the artifacts described in the Rational Unified Process are similarly incorporated into the process.

In APCS, we have used notions such as the System View as a generic means to indicate the existence of inputs and outputs to process steps as well as the kind of data and knowledge that is necessary to those steps. How this data and knowledge are aggregated into specific document or artifact types is particular to organizational constraints. We believe, however, that it should not be the case that any artifact required by an instantiated process will invalidate the generic process steps defined in the IDEF figures.

## 5.2 Further Work

The process defined by APCS represents only one portion of the activities carried out in developing a COTS-based system. There is another domain of activity that parallels APCS, namely, the activities in the sphere of project management. We have alluded to these management activities, and have termed them Executive activities. We also noted that, from an IDEF0 perspective, they would be decomposed starting from the first and third boxes in Figure 6.

The relationships between management activities and development activities are complex in any circumstance, whether a COTS-based development or otherwise. Management activities will interact with development activities; they will provide inputs to and receive output from them (e.g., decisions based on evaluation of products, decisions concerning the time allocated to integration).

However, just as we indicated in describing the development activities themselves, we believe that the demands of a COTS-based project mean that there will be many changes, both broad and subtle, that must be made to traditional modes of management. Similarly, the use of a spiral approach will also impose changes on traditional management activities. Where a traditional management approach might be predicated on the existence of a well-defined requirements specification, a COTS-based management approach must make allowance for a specification with a very sparse notion of requirements during the initial iterations. Where a traditional management process might depend on a clear delineation between development responsibilities and maintenance responsibilities, this distinction may well become more fluid in a COTS-based circumstance.

We have not attempted, in APCS, to grapple with these questions; the development issues were complex enough in their own right. However, it will be necessary that the question of a COTS-based management process be addressed at some point in the future, and its relation to APCS defined.

# References/Bibliography

| | |
|---|---|
| **[Albert 02]** | Albert, C. & Brownsword, L. *Evolutionary Process for Integrating COTS-Based Systems (EPIC)* (CMU/SEI-2002-TR-005, ADA408653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu /publications/documents /02.reports/02tr005.html>. |
| **[Boehm 88]** | Boehm, B. "A Spiral Model of Software Development and Enhancement" *Computer 21*, 5 (May 1988): 61-72. |
| **[Boehm 99]** | Boehm, B. & Port, D. "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them." *ACM Software Engineering Notes 24*, 1 (January 1999): 36-48. |
| **[Carney 01]** | Carney, D.; Hissam, S.; & Plakosh, D. "Complex COTS-Based Software Systems: Practical Steps for Their Maintenance." *Journal of Software Maintenance: Research and Practice 12,* 6 (December 2000): 357-376. |
| **[Clinger-Cohen 96]** | Clinger-Cohen Act. Section 808, Omnibus Consolidated Appropriations Act (Pub. L 104-208): Division D (Federal Acquisition Reform Act of 1996), Division E (Information Technology Management Reform Act of 1996) (Clinger-Cohen Act of 1996). Washington, DC: 104th Congress of the United States, 1996. |
| **[Hissam 98]** | Hissam, S. & Carney, D. "Isolating Faults in Complex COTS-Based Systems." *Journal of Software Maintenance: Research and Practice 11*,3 (March 1999): 183-199. |
| **[Kruchten 00]** | Kruchten, P. *The Rational Unified Process: An Introduction.* Reading, MA: Addison-Wesley, 2000. |

| | |
|---|---|
| **[Marca 88]** | Marca, D. *SADT: Structured Analysis and Design Technique.* New York, NY: McGraw-Hill, 1988. |
| **[PSESWG 93]** | Project Support Environmental Standards Working Group. Oberndorf, P.; Brown, A.; Carney, D. & Zelkowitz, M., eds. *Reference Model for Project Support Environments Version 1.0.* Warminster PA: Naval Air Warfare Center, Air Craft Division Warminster, 1993. |
| **[Royce 70]** | Royce, W. "Managing the Development of Large Software Systems: Concepts and Techniques." *WESCON Technical Papers 14, Western Electronic Show and Convention,* Los Angeles, CA, Aug. 1970. |
| **[Wallnau 01]** | Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*. Boston, MA: Addison-Wesley, 2001. |

| REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188* |
|---|---|

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| | | |
|---|---|---|
| 1. AGENCY USE ONLY<br>(Leave Blank) | 2. REPORT DATE<br>May 2003 | 3. REPORT TYPE AND DATES COVERED<br>Final |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>A Basis for an Assembly Process for COTS-Based Systems (APCS) | 5. FUNDING NUMBERS<br>F19628-00-C-0003 |
| 6. AUTHOR(S)<br>David J. Carney, Patrick R.H. Place, Patricia A. Oberndorf | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Software Engineering Institute<br>Carnegie Mellon University<br>Pittsburgh, PA 15213 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>CMU/SEI-2003-TR-010 |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>HQ ESC/XPK<br>5 Eglin Street<br>Hanscom AFB, MA 01731-2116 | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER<br>ESC-TR-2003-010 |
| 11. SUPPLEMENTARY NOTES | |
| 12A DISTRIBUTION/AVAILABILITY STATEMENT<br>Unclassified/Unlimited, DTIC, NTIS | 12B DISTRIBUTION CODE |

13. ABSTRACT (MAXIMUM 200 WORDS)

This paper describes a generic process framework for developing software systems based on commercial off-the-shelf (COTS) products. The framework is based on Barry Boehm's familiar spiral development process. However, it is primarily intended for projects that make significant use of commercial components and other pre-existing software as elements of the system to be fielded. The aspects of the process that are most affected by this reliance on COTS components lie in the area of requirements, and the description of the process is most extensive in that area. The necessity of using system prototypes as the major vehicle for reducing risk is assumed, as are parallel and interleaved periods of gathering and refining knowledge about the system to be built. Each element of the process is first described and then depicted in several models, using Integrated Definition modeling technique (IDEF0). The paper describes how the interactions between the candidate COTS components, the stakeholders' implicit and explicit needs, and the context in which the system will operate all provide interacting constraints on both the process and the resulting system.

| | |
|---|---|
| 14. SUBJECT TERMS<br>commercial off-the-shelf, COTS, process, evolutionary process, Rational Unified Process, RUP, spiral process, EPIC, inception, elaboration, construction, transition, phase, iteration | 15. NUMBER OF PAGES<br>43 |
| 16. PRICE CODE | |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>Unclassified | 20. LIMITATION OF ABSTRACT<br>UL |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102